THE OFFICE OF ENTERPRISE TECHNOLOGY STRATEGIES

Statewide Technical Architecture

Implementation Guideline:

Application Development Guidelines for Java 2 Platform, Enterprise Edition

STATEWIDE TECHNICAL ARCHITECTURE

Implementation Guideline: Application Development Guidelines for Java 2 Platform, Enterprise Edition

Revised Date:		Version:	1.0.0
Revision Approved Date:			
Date of Last Review:	March 1, 2004		
Date Retired:			
Architecture Interdependencies:			
Reviewer Notes: Initial release.			

Table of Contents

List of Figures

Introduction

The intent of this document is to outline the implementation guidelines that the State of North Carolina has adopted to ensure uniform and consistent implementations of Java 2 Enterprise Edition (J2EE) Platform based solutions throughout the enterprise.

Detailed aspects of Java development will not be covered in this document. There are resources listed in the following section that provide mechanisms for gaining a greater depth and knowledge about the Java programming language and the J2EE platform. The information contained in this document is a summarization of key lessons learned, extensive research, industry best practices, and analysis of the Java BluePrints[1] program created by Sun Microsystems. While the Java BluePrints program details many of the possibilities within the J2EE platform, this document provides approaches to designing and developing J2EE solutions that comply with the North Carolina Statewide Technical Architecture (NCSTA). For additional information on the NCSTA and compliant technologies, please contact the Enterprise Technology Strategies (ETS) office at http://www.ncsta.gov.

Guidelines within this document provide solution developers with resources to help create well-designed J2EE solutions that have are flexible, scalable, and extensible, a fundamental principle of the NCSTA. The focus will be on application architecture, such as distributing J2EE application functionality across tiers and choosing among design options within each tier. This document assumes that the reader already has basic knowledge of the J2EE platform. For more information about Java and J2EE refer to the resources listed in the Support Mechanisms section.

This document in no way indicates or implies a preference in a vendor or technology by the State of North Carolina. It is provided to ensure that agencies implementing J2EE solutions have the resources necessary to successfully deploy their systems in the most advantageous manner to both the agency and the state.

Document Structure

This guide uses a set of terms such as presentation, business logic, integration, and data to represent the physical and logical tiers of abstract application architecture. These terms relate to the principles, practices, and standards provided in the NCSTA. This guide also uses a set of terms to represent an application's design from a J2EE perspective. Many of the J2EE Platform specific terms have a direct correlation to the abstract terms, while others such as Enterprise Information Systems (EIS) represent an aggregation of elements and services. Listed are the major abstract terms along with the document sections that discuss each topic.

- **Presentation** - the ***presentation*** of the user interface and the processes that control the users actions are discussed in both The Client Tier and The Web Tier sections.

[1] Designing Enterprise Applications with the J2EE Platform, Second Edition - 2002 Sun Microsystems, Inc. http://java.sun.com/blueprints/

- **Business Logic** - details about the application-specific ***business logic***, to include the system-level services such as transaction management, concurrency control, and data access are discussed in The Enterprise Bean Tier section.
- **Integration and Data** - the components used to ***integrate*** applications with existing enterprise information systems, other legacy applications, and provide the ***data*** critical to the business processes are defined in The Enterprise Bean Tier section.

For a list of other terms and definitions used within this document please refer to the Definition of Terms section of the guide.

Support Mechanisms

The creation of applications using the J2EE platform is complex and requires an in-depth knowledge and diverse skill set in areas such as application design, application programming, and security, as well as a general understanding of networking and database technologies. To address this complexity, there are a number of support mechanisms that the State of North Carolina can leverage when developing applications using the J2EE Platform.

- **J2EE Vendor Account Executives (State of North Carolina)** - The State of North Carolina can engaged a number of vendors for help in developing J2EE applications. Vendor account executives should be viewed as key strategic resources for developing and deploying applications based on the J2EE Platform for their specific products. They focus on fostering a positive relationship between the vendor community and the State; and can leverage resources within their organization as well as partners for guidance and support for J2EE development efforts. For specific contact information and help in facilitating communication with state vendors please contact the ETS office.
    - **Email -** ETS@ncmail.net **Phone -** (919) 981-5510

- **Knowledge base or online resources for the Java community** – The links provided are meant to provide developers, with the tools, technologies, and expertise needed to maximize productivity with the J2EE Platform.
    - **Java Developer Connection –** http://java.sun.com/
    - **Java Technology Collaboration** - www.java.net
    - **Community Development of Java Technology Specification -** www.jcp.org
    - **The Server Side.com** - www.theserverside.com
    - **Java Center of Excellence -** http://www.sun.com/service/solutions/java/index.html
    - **Developer Works Java Technology -** http://www.ibm.com/developerworks/java/
    - **Java Developer Center** - http://otn.oracle.com/tech/java/index.html
    - **Dev2Dev -** http://dev2dev.bea.com/index.jsp

- **Java User Groups** – A Java User Group is a group of people who share a common interest in Java technology and meet on a regular basis to share technical ideas and information. The actual structure of a JUG can vary greatly - from a small number of friends and coworkers meeting informally in the evening, to a large group of companies based in the same geographic area.
    - **Java User's Group Listing -** http://java.sun.com/jugs/
    - **Triangle (NC) Java User's Groups -** http://trijug.org/

Additional Links and Resources

1. The Java BluePrints Web site http://java.sun.com/blueprints
2. The Java Tutorial Web site http://java.sun.com/docs/books/tutorial
3. Java Web Start Web site http://java.sun.com/products/javawebstart
4. Java Authentication and Authorization Service Web site http://java.sun.com/products/jaas
5. J2EE Platform Specification http://java.sun.com/j2ee
6. The J2EE Tutorial. S. Bodoff, D. Green, K. Haase, E. Jendrock, M. Pawlan. Copyright 2001, Sun Microsystems, Inc. http://java.sun.com/j2ee/tutorial/index.html
7. The JFC/Swing Tutorial. M. Campione, K. Walrath. Copyright 2000, Addison-Wesley. Also available as http://java.sun.com/docs/books/tutorial/uiswing/index.html
8. The Java Tutorial, Third Edition: A Short Course on the Basics. M. Campione, K. Walrath, A. Huml. Copyright 2000, Addison-Wesley. Also available as http://java.sun.com/docs/books/tutorial/index.html
9. Java Technology and XML. T. Violleau. Copyright 2001, Sun Microsystems, Inc. http://developer.java.sun.com/developer/technicalArticles/xml/JavaTechandXML/
10. Enterprise JavaBeans Specification http://java.sun.com/products/ejb
11. The J2EE Connector Specification http://java.sun.com/products/j2ee
12. JDBC API, (JDBC specification) http://java.sun.com/products/jdbc
13. JDBC Standard Extension API (JDBC extension specification) http://java.sun.com/products/jdbc
14. Java Message Service (JMS Specification) http://java.sun.com/products/jms
15. The JavaTM Servlet Specification http://jcp.org/aboutJava/communityprocess/first/jsr053/index.html
16. The Java Transaction Architecture specification http://java.sun.com/products/jta
17. Web Services http://java.sun.com/webservices/
18. Web Service Solutions http://www.sun.com/service/solutions/java/

J2EE Platform Technologies

The J2EE platform specifies technologies to support multi-tier applications. These technologies fall into three categories: component, service, and communication.

The component technologies are those used by developers to create the essential parts of the application, namely the user interface and the business logic. The component technologies allow the development of modules that can be reused by multiple applications.

Since most applications require access to existing information systems, the J2EE platform supports Application Programming Interfaces (API) that provide access to databases, information systems such as SAP and CICS, and services such as transaction, naming and directory, and asynchronous communication. Finally, the J2EE platform provides technologies that enable communication between clients and servers and between collaborating objects hosted by different servers.

Component Technologies

A *component* is an application-level software unit. In addition to JavaBeans components, which are part of the Java 2 Standard Edition (J2SE) platform, the J2EE platform supports the following types of components: clients, Enterprise JavaBeans (EJB) components, web components, and resource adapter components. Application clients run on a client platform, while EJB, web, and resource adapter components run on a server platform as a separate tier from the client platform tier.

All J2EE components depend on the runtime support of a system-level entity called a *container*. Containers provide components with services such as lifecycle management, security, deployment, and threading. Because containers manage these services, many component behaviors can be declaratively customized when the component is deployed in the container.

Clients

The J2EE platform allows different types of clients to interact with server-side components.

- An *application client* executes in its own client container. (The client container is a set of libraries and APIs that support the client code). Application clients are user interface programs that can directly interact with the EJB tier of a J2EE platform-based application using the Remote Method Invocation-Internet Inter-ORB Protocol (RMI-IIOP). These clients have full access to J2EE platform services such as Java Naming and Directory Interface (JNDI) lookups, asynchronous messaging, and the Java Database Connectivity (JDBC) API. An application's client container provides access to these J2EE services and handles RMI-IIOP communication.
- A *Web Start-enabled rich client* is a based on JFC/Swing APIs and enabled for the J2EE platform through the Java Web Start technology. A rich client has increased user interface features available to it, such as a better interactive environment and richer graphic capabilities, along with the J2EE platform features and services. Best practices should still be followed with client-side components by keeping the application's footprint minimal and business rules hosted by the application server. Java Web Start technology enables

application deployment through a single-step download-and-launch process performed by means of a web browser. Rich clients communicate with the server using the J2SE environment to execute XML over HTTP(S). As Web Services gain ground in the future, these rich clients are well positioned to efficiently use open communication standards such as JAX-RPC technology.

- A *wireless client* is based on Mobile Information Device Profile (MIDP) technology. MIDP is a set of Java APIs, which, along with Connected Limited Device Configuration (CLDC), provides a complete Java 2 Micro Edition (J2ME) environment for wireless devices.

EJB Components

The EJB architecture is a server-side technology for developing and deploying components containing the business logic of an enterprise application. Enterprise JavaBeans components, also referred to as enterprise beans, are scalable, transactional, and multi-user secure. There are three types of enterprise beans: session beans, entity beans, and message-driven beans. Session and entity beans have both a component interface and a home interface. The home interface defines methods to create, find, remove, and access metadata for the bean. The component interfaces define the bean’s business logic methods. Message-driven beans do not have component and home interfaces.

An enterprise bean’s component and home interfaces are required to be either local or remote. Remote interfaces utilize the Remote Method Invocation (RMI) technology, which provides location independence to the beans clients. Regardless of whether the client of a bean that implements a remote interface is located on the same Virtual Machine (VM) or a different VM, the client uses the same API to access the bean’s methods. Arguments and return results are passed by value between a client and a remote enterprise bean, however this generates serialization overhead.

A client of an enterprise bean that implements a local interface must be located in the same VM as the bean. Because object arguments and return results are passed by reference between a client and a local enterprise bean, the serialization overhead is removed. However, utilizing local interfaces limits the application’s ability to scale. Because the presentation and business logic must be collocated within the same JVM, scalability is limited to a single server. Additionally, components that implement an EJB local interface cannot be instantiated from a distributed system. Code reuse and application integration will be limited.

An EJB container hosts the enterprise beans components. In addition to standard container services, an EJB container provides a range of transaction and persistence services and access to the J2EE service and communication APIs. An EJB container is usually provided as part of a vendor’s application server offering.

Web Components

A *web component* is a software entity that provides a response to a request. A web component typically generates the user interface for a web-based application. The J2EE platform specifies two types of web components: servlets and Java Server Pages (JSP) pages.

A *servlet* is a component that extends the functionality of a web server. A web server hosts Java servlet classes that execute within a servlet container. The web server maps a set of URLs to a servlet so that HTTP requests to these URLs invoke the mapped servlet. When a servlet receives a request from a client, it generates a response by invoking business logic in enterprise beans. It then sends the response—as an HTML or XML document—to the requestor.

To comply with the NCSTA, presentation components (servlets & JSPs) must not access the application's data directly. Data access must be managed by the application's business logic.

The JavaServer Pages (JSP) technology provides an extensible way to generate dynamic content for a web client. A JSP page is a text-based document that describes how to process a request to create a response. Most JSP pages use JavaBeans and/or Enterprise JavaBeans components to perform the more complex processing required of the application. Standard JSP actions can access and instantiate beans, set or retrieve bean attributes, and download applets. JSP technology is extensible through the development of custom actions, or tags, which are encapsulated in tag libraries.

Servlet containers, JSP containers, and web containers host web components. In addition to standard container services, a *servlet container* provides the network services by which requests and responses are sent. It also decodes requests and formats responses. All servlet containers must support HTTP as a protocol for requests and responses; they may also support other request-response protocols such as HTTPS. A *JSP container* provides the same services as a servlet container. Servlet and JSP containers are collectively referred to as *web containers*.

Service Technologies

The J2EE platform service technologies allow applications to access a wide range of services in a uniform manner. This section describes technologies that provide access to databases, transactions, XML processing, naming and directory services, and enterprise information systems.

JDBC

The JDBC API provides database-independent connectivity between the J2EE platform and a wide range of tabular data sources. JDBC technology allows an application component provider to:

- Perform connection and authentication to a database server
- Manage transactions
- Move SQL statements to a database engine for preprocessing and execution
- Execute stored procedures
- Inspect and modify the results from **`Select`** statements

The J2EE platform requires both the JDBC Core API (included in the J2SE platform), and the JDBC Extension API, which provides row sets, connection naming via JNDI, connection pooling, and distributed transaction support. The connection pooling and distributed transaction features are intended for use by JDBC drivers to coordinate with a J2EE server.

Transactions

The Java Transaction API (JTA) allows applications to access transactions in a manner that is independent of specific implementations. JTA specifies standard Java interfaces between a transaction manager and the parties involved in a distributed transaction system: the transactional application, the J2EE server, and the manager that controls access to the shared resources affected by the transactions.

The Java Transaction Service (JTS) specifies the implementation of a transaction manager that supports JTA and implements the Java mapping of the Object Management Group Object Transaction Service specification. A JTS transaction manager provides the services and management functions required to support transaction demarcation, transactional resource management, synchronization, and propagation of information that is specific to a particular transaction instance.

Naming and Directory Interface

The Java Naming and Directory Interface (JNDI) API provides naming and directory functionality. It provides applications with methods for performing standard directory operations, such as associating attributes with objects and searching for objects using their attributes. Using JNDI, an application can store and retrieve any type of named Java object.

Because JNDI is independent of any specific implementations, applications can use JNDI to access multiple naming and directory services, including existing naming and directory services such as LDAP, NDS, DNS, and NIS. This allows applications to coexist with legacy applications and systems.

Connector Architecture

The J2EE Connector architecture is a standard API for connecting the J2EE platform to Enterprise Information Systems (EIS), such as Enterprise Resource planning (ERP), mainframe transaction processing, and backend database systems. The architecture addresses the issues involved when integrating existing enterprise information systems, such as SAP, CICS, legacy applications, and non-relational databases, with an EJB server and enterprise applications. The J2EE Connector architecture defines a set of scalable, secure, and transactional mechanisms for integrating an EIS with a J2EE platform.

XML Parsers

The Java API for XML Processing (JAXP) technology supports the processing of XML documents using Document Object Model (DOM), Simple API for XML (SAX), and eXtensible Stylesheet Language Transformations (XSLT). JAXP enables applications to parse and transform XML documents independent of a particular XML processing implementation. Depending on the needs of the application, developers have the flexibility to swap between XML processors, such as between high-performance or memory-conservative parsers, with no application code changes.

Communication Technologies

Communication technologies provide mechanisms for communication between clients and servers and between collaborating objects hosted by different servers. This section provides an overview of Internet protocols, remote method invocation protocols, Object Management Group protocols, and messaging technologies provided by the J2EE Platform.

Internet Protocols

Internet protocols define the standards by which the different pieces of the J2EE platform communicate with each other and with remote entities. The J2EE platform supports the following Internet protocols: TCP/IP, HTTP, and SSL (refer to the Definition of Terms section).

RMI

Remote Method Invocation (RMI) is a set of APIs that allow developers to build distributed applications in the Java programming language. RMI uses Java language interfaces to define remote objects and a combination of Java serialization technology and the Java Remote Method Protocol (JRMP) to turn local method invocations into remote method invocations. The J2EE platform supports the JRMP protocol, the transport mechanism for communication between objects in the Java language in different address spaces.

OMG

Object Management Group (OMG) protocols allow objects hosted by the J2EE platform to access remote objects developed using the OMG's Common Object Request Broker Architecture (CORBA) technologies and vice versa. CORBA objects are defined using the Interface Definition Language (IDL). An application component provider defines the interface of a remote object in IDL and then uses an IDL compiler to generate client and server stubs that connect object implementations to an Object Request Broker (ORB), a library that enables CORBA objects to locate and communicate with one another. ORBs communicate with each other using the Internet Inter-ORB Protocol (IIOP). The OMG technologies required by the J2EE platform are Java IDL and RMI-IIOP.

JMS

The Java Message Service (JMS) API allows J2EE applications to access enterprise messaging systems such as IBM MQ Series, Microsoft's MSMQ, and TIBCO Rendezvous. JMS messages contain well-defined information that describes specific business actions. Through the exchange of these messages, applications track the progress of enterprise activities. The JMS API supports both point-to-point and publish-subscribe styles of messaging.

In *point-to-point messaging*, a client sends a message to the message queue of another client. Often a client will have all its messages delivered to a single queue. Most queues are created administratively and are treated as static resources by their clients.

In *publish-subscribe messaging*, clients publish messages to and subscribe to messages from well-known nodes in a content-based hierarchy called *topics*. A topic can be thought of as a message broker that gathers and distributes messages addressed to it. By relying on the topic as an intermediary, message publishers are independent of subscribers and vice-versa. The topic automatically adapts as both publishers and subscribers come and go.

Reference Architecture Design

The architecture of any enterprise application can be thought of in two different ways: Physical View and the Logical View. The Logical View addresses entities (components) that perform a certain function in the overall architecture - it is platform agnostic. The logical components that make up the Logical View map to the real world platform on which they live, gives rise to the Physical View. Both views will be presented in this document.

Physical View

Figure 1 illustrates a typical high-level Physical Design for application architecture. It should be noted that this architecture is offered as a best practice and will sometimes need to be implemented in a somewhat different manner than is illustrated below. This is to be expected, however, any significant deviation from the NCSTA, such as locating anything other than the presentation-tier in front of the firewall, must have strong business justification to warrant approval by ETS. The following annotated diagram is described below. Additional details and examples can be found in the position and white papers section of the ETS technical architecture website (http://www.ncsta.gov).

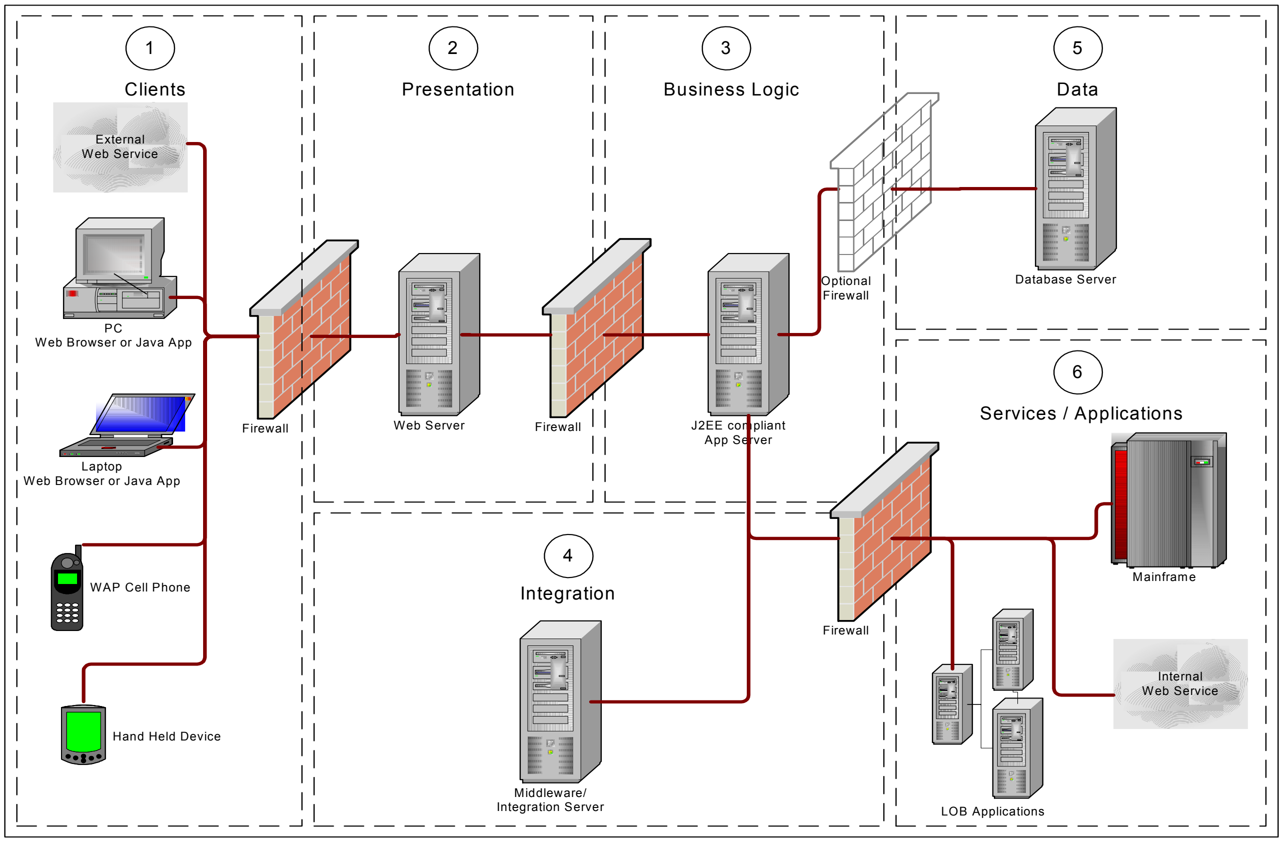

Figure 1: Physical View of Common Architecture

1. **Clients:** The Architecture should be able to support the widest variety of clients. These include:
   - Standard PC browsers on laptops, desktops, workstations, and tablets.
   - Mobile Devices such as Pocket PC, Palm, WAP enabled phones, etc.
   - Web Services that need access to information provided by the application.
2. **Presentation:** The presentation tier provides an interface to clients and contains the components on the server side that are directly responsible for generating web pages. In particular, an application utilizing J2EE usually makes extensive use of Java Server Pages (JSP), servlet, and static HTML/XML technologies within the presentation tier.
3. **Business Logic:** The presentation tier makes requests for services through the use of Enterprise JavaBeans (EJB) running in the business logic tier. The beans make use of system services provided by the middle-tier application server to access and manage the data tier. EJBs are reusable components that can be accessed by multiple client programs including other application components. The components are deployed onto a J2EE compliant application server.
4. **Integration:** Integration servers usually support a suite of business integration tools for integrating and coordinating heterogeneous applications. The Integration Server's tools and software components integrate newly developed applications, legacy applications, and commercial-off-the-shelf (COTS) packages into business processes that are managed and controlled by a process engine. The technologies consist of application messaging, the connection architecture, and web services.
5. **Data:** Relational database management systems (RDBMS) are the most prevalent approach to data storage and considered an industry best practice. Many application component providers use the JDBC API for accessing relational databases to manage persistent data for their applications. In addition to the points outlined below, the NCSTA mandates that new databases must be relational (RDBMS) and ANSI-SQL compliant.

   Stored Procedures:

   To retain database portability the use of stored procedures for business logic is not compliant with requirements defined in the NCSTA. Stored procedures may only be used when there is a viable design goal that can only be solved by them or for database specific tasks. Otherwise, SQL statements for data access should be managed within the business logic tier.

   Database Security:

   As maintained in the NCSTA all applications should adhere to the state's Identity and Access Management Service (IAMS) authentication model for database security. IAMS will complement the existing database security using a role-based access control methodology and provide cross platform integration for users.

   Additional principles, practices, and standards regarding compliance with the NCSTA can be found in the Data Domain located at http://www.ncsta.gov. The programming design points concerning the data tier will be addressed later in this document.
6. **Services / Applications:** Sometimes a distributed application will need to access legacy systems, LOB applications, and/or other services (e.g. web services). To achieve this an existing or new

middleware/EAI solution may be leveraged. This is provided that it complies with the principles, practices, and standards for system integration set forth in the NCSTA.

Logical View

The following section will address the architectural design considerations of the Logical View. This section is targeted more toward developers and architects. Providing them with a basic overview of the different application scenarios available by the J2EE platform and compliant with the NCSTA.

The J2EE specifications encourage architectural diversity. The J2EE specifications and technologies make few assumptions about the details of API implementations. The application-level decisions and choices are ultimately a trade-off between functional richness and complexity.

The J2EE programming model is flexible enough for applications that support a variety of client types, with both the web container and EJB container as optional. Figure 2 reflects a range of possible application configurations, including cases where clients interact solely with the web container, where clients interact directly with the EJB container, and full-blown multi-tier applications with stand-alone clients, web-tier components, middle-tier EJB components, and EIS-tier access to resources and data.

While the J2EE platform has no implicit bias favoring one application scenario over another, applications developed for the State of North Carolina must adhere to the principles, practices, and standards set forth in the NCSTA.

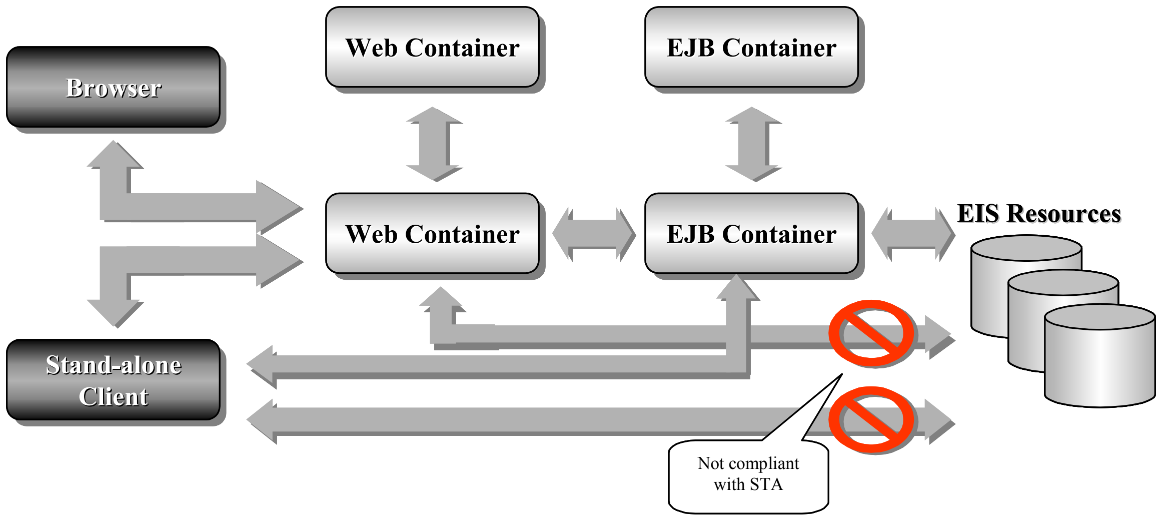

Figure 2: J2EE application scenarios

In order to be compliant with the principles, practices, and standards contained in the NCSTA, applications designed for North Carolina state agencies must utilize a 3/N-tier architecture that uses both a web container and an EJB container (Stand-alone applications may only utilize an EJB Container). By designing systems in this way, the State of North Carolina gains the following benefits:

- The ability to make changes to the "look" of the application without affecting existing backend services.
- The ability to partition the application along the lines of presentation and business logic in order to increase modularity.
- The ability to simplify the development process, by allowing work to proceed along relatively independent but cooperating tracks.
- The ability to have developers familiar with back-office applications unburdened from GUI and graphic design work, for which they may not be ideally qualified.
- The ability to have the necessary vocabulary to communicate the business logic to teams concerned with human factors and the aesthetics of the application.
- The ability to assemble back-office applications using components from a variety of sources, including off-the-shelf business logic components.
- The ability to deploy transactional components across multiple hardware and software platforms independently of the underlying database technology.
- The ability to externalize internal data through standard interfaces without having to make many assumptions about the consumer of the data and to accomplish this in a loosely coupled manner.

Multi-tier Web Applications

Figure 3 illustrates an application scenario in which the web container hosts web components that are almost exclusively dedicated to handling a given application's presentation logic. JSP pages, supported by servlets, generate dynamic web content for delivery to the client. The EJB container hosts application components that use EIS resources (e.g. databases, legacy systems) to service requests from the web Container tier components. This architecture decouples data access from the application's user interface and encapsulates the business logic into a single location. The architecture is also implicitly scalable. Application back-office functionality is relatively isolated from the end-user look and feel.

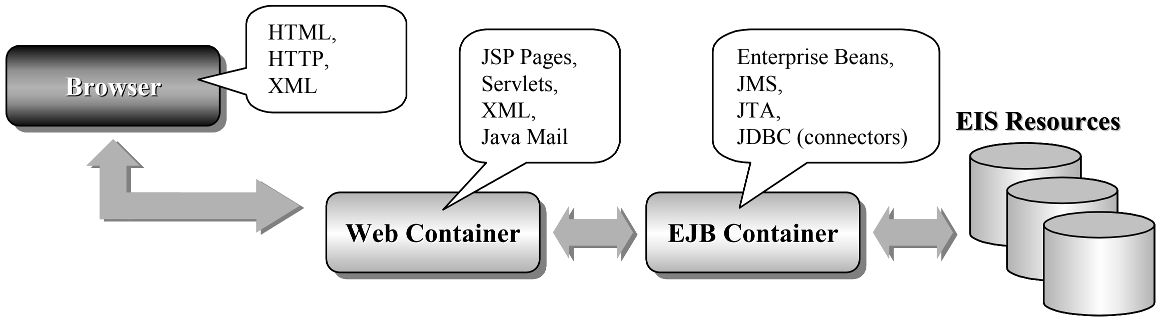
Browser
HTML, HTTP, XML
JSP Pages, Servlets, XML, Java Mail
Enterprise Beans, JMS, JTA, JDBC (connectors)
EIS Resources
Web Container
EJB Container

Figure 3: Multi-tier web application

In the web-tier, the decision of whether to use JSP pages or servlets for presentation components should be based on the following:

- JSP technology is intended for application user interface components.
- Java Servlets are preferred for request processing and application control logic.

Servlets and JSP pages work together to provide dynamic content from the web-tier.

Stand-alone Client Applications

Figure 4 illustrates a stand-alone client scenario.

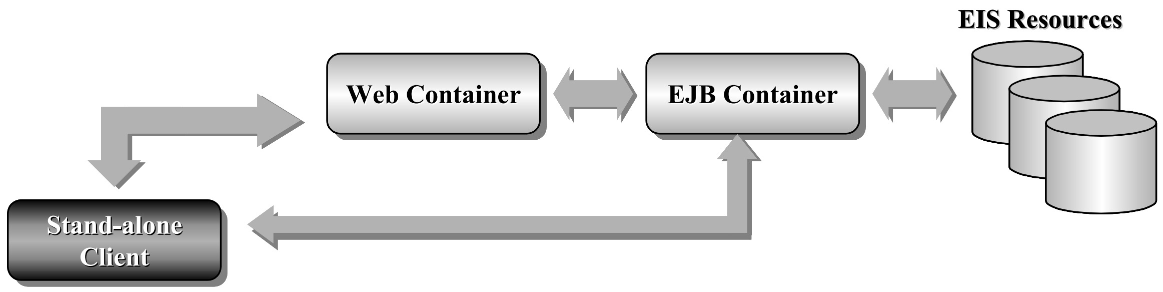

Figure 4: Stand-alone client application

Within the stand-alone client scenario, two NCSTA compliant configurations exist:

1. Stand-alone clients (EJB clients) interacting directly with enterprise beans hosted in an EJB container within an EJB server, as shown in Figure 5. This scenario uses RMI-IIOP, and the EJB server accesses EIS resources (e.g. databases, legacy applications) using JDBC and the J2EE Connector architecture.

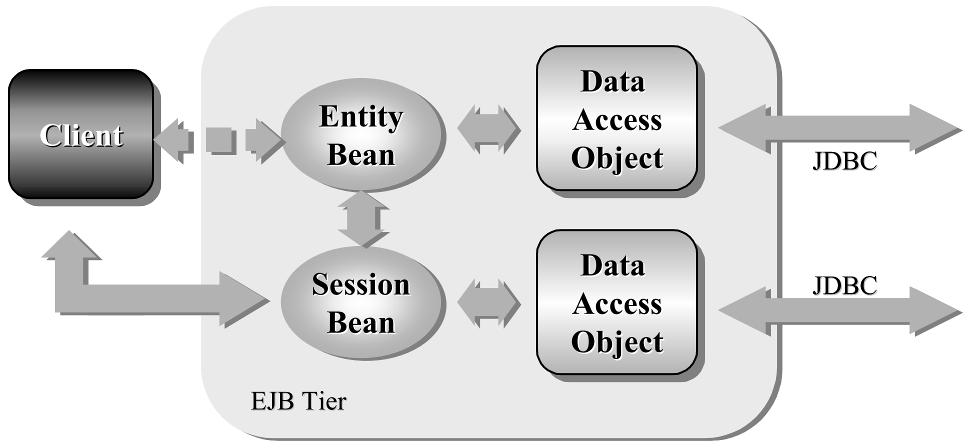

Figure 5: EJB-centric Java client

2. Stand-alone clients implemented in the Java language or another programming language, consuming dynamic web content (usually XML data messages). In this scenario, the web container essentially handles XML transformations (XSLT) and provides web connectivity to clients. Presentation logic occurs in the client tier (client desktop and HTTP server). The

EJB container handles business logic via enterprise beans and access to EIS resources (e.g. databases).

With either implementation, the NCSTA states that application components must maintain clear separation between the business and presentation logic. Stand-alone clients should only include minimal field validation and process flow, all business logic must be implemented within server-side enterprise beans.

Service / Integration Applications

Figure 6 illustrates a service/integration scenario. This scenario focuses on peer-level interactions between both web and EJB containers. The J2EE programming model promotes the use of XML data messaging over HTTP as the primary means of establishing loosely coupled communications between web containers.

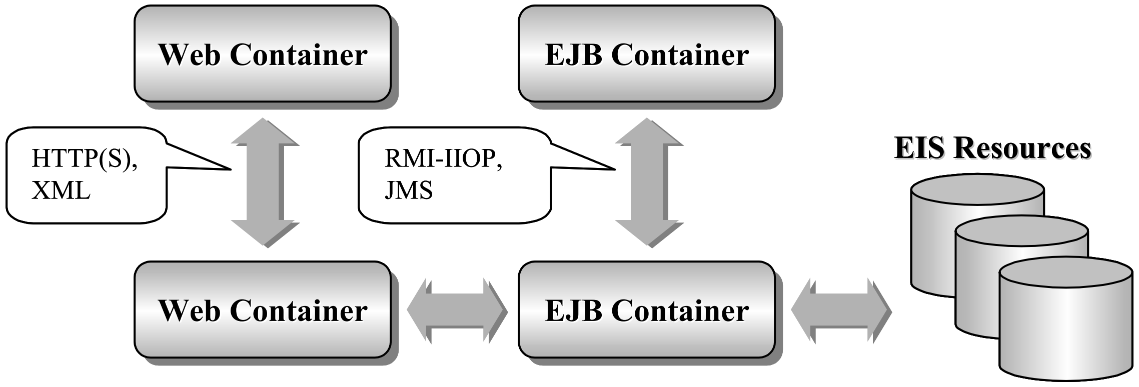

Figure 6: Service / Integration application

Future releases of the J2EE platform will provide additional functionality in the form of Java APIs for XML, which enable more complete support for loosely coupled applications through XML-based Web Services.

The Client Tier

From a developer's point of view, a J2EE application can support many types of clients. J2EE clients can run on laptops, desktops, palmtops, and cell phones. They can connect from within an enterprise's intranet or across the World Wide Web, through a wired network or a wireless network or a combination of both. They can range from browser-based to a rich, standalone client. The NCSTA recommends a thin client that interacts with a set of services provided through the implementation of a Service Oriented Architecture (SOA).

From a user's point of view, the client *is* the application. It must be useful, usable, and responsive. Because the user places high expectations on the client, you must choose your client strategy carefully, making sure to consider both technical forces (such as the network) and non-technical forces (such as the nature of the application).

While the J2EE platform encourages thin-client architectures, J2EE clients are still responsible for providing the following capabilities:

- **Presenting the user interface**—Although a client presents the views to a user, the logic for the views may be programmed on the client (standalone) or downloaded from a server.
- **Validating user inputs**—Although the EIS and EJB tier must enforce constraints on model data (since they contain the data), a client may also enforce data constraints by validating user inputs. Client-side validation should be minimized to limit the amount of code that is transferred across the network.
- **Communicating with the server**—When a user requests functionality that resides on a server, the user's client must present that request to the server using a standard protocol.
- **Managing conversational state**—Applications need to track information as a user goes through a workflow or process (effectively conversing with the application). The client may track none, some, or all of this information, known as conversational state. *Refer to the specific NCSTA security standards for capturing application state.*

The next two sections consider browser clients and stand-alone Java clients separately. You do not have to pick one or the other; a J2EE application can accommodate both browser and Java clients. The web-tier section goes into further detail on how to design the web-tier to support multiple types of clients.

Browser Clients

Browsers are the thinnest of clients; they display data to their users and rely on servers for application functionality. From a deployment perspective, browser clients are attractive for a couple of reasons. First, browser-based applications require minimal client-side updating. When an application changes, server-side code is redeployed, but browsers are almost always unaffected. Second, they are ubiquitous. Almost every computer has a web browser and many mobile devices have a micro-browser.

Browser clients download documents from a server. These documents contain data as well as instructions for presenting that data. The documents are usually dynamically generated by JSP pages (and less often by Java servlets) and written in a presentational markup language such as Hypertext Markup Language (HTML). A presentational markup language allows a single document to have a reasonable presentation regardless of the browser that presents it.

Browsers have a few strengths that make them viable enterprise application clients. First, they offer a familiar environment. Browsers are widely deployed and used, and the interactions they offer are fairly standard. This makes browsers popular, particularly with novice users. Second, browser clients can be easy to implement. The markup languages that browsers use provide high-level abstractions for how data is presented, leaving the mechanics of presentation and event handling to the browser. Finally, users can be presented with similar look-and-feel for a suite of applications through the use of common tag libraries and style sheets.

Stand-alone Clients

Although a Java client contains an application's user interface, the presentation logic behind this interface may be located on a server, as it would be for a browser-based application, or it may be programmed from the ground up on the client.

Both browser-based and standalone clients must provide a clear separation between the presentation and business logic to be compliant with the NCSTA. A standalone client would still access server-side business logic.

Implementing the user interface for a Java client usually requires more effort to implement than a browser interface, but there are some benefits for doing so if the business problem calls for such an implementation. First, Java client interfaces offer a richer user experience; with programmable GUI components, you can create more natural interfaces for the task at hand. Second, and perhaps more importantly, full programmability makes Java clients much more responsive than browser interfaces.

When a Java client and a browser client request the same data, the Java client consumes less bandwidth. For example, when a browser requests a list of orders and a Java client requests the same list, the response is larger for the browser because it includes presentation logic. The Java client, on the other hand, gets the data and nothing more.

When using browser clients, you must write code for translating user actions into HTTP requests, HTTP requests into application events, event responses into HTTP responses, and HTTP responses into view updates. On the other hand, when using EJB clients, you do not need to write such code because the clients connect directly to the EJB tier using Java Remote Method Invocation (RMI) calls (see Figure 5).

The Web Tier

A J2EE application's web-tier (referred to as the presentation-tier by the NCSTA) makes the application's business logic available to distributed clients. The web-tier handles all of a J2EE application's communication with web clients, invoking business logic and transmitting data in response to incoming requests.

A server in the web-tier processes HTTP requests. In a J2EE application, the web-tier usually manages the interaction between web clients and the application's business logic. The web-tier typically produces HTML or XML content, though the web-tier can generate and serve virtually any content type.

The web-tier typically performs the following functions in a J2EE application:

- **Web-enables business logic**—The web-tier manages interaction between web clients and application business logic, which should be placed within the business logic tier.
- **Generates dynamic content**—web-tier components generate content dynamically, in entirely arbitrary data formats, including HTML, images, sound, and video.
- **Presents data and collects input**—web-tier components translate HTTP PUT and GET actions into a form that the business logic understands and present results as web content.
- **Controls screen flow**—The logic that determines which "screen" (that is, which page) to display next usually resides in the web-tier, because screen flow tends to be specific to client capabilities.
- **Maintains state**—The web-tier has a simple, flexible mechanism for accumulating data for transactions and for interaction context over the lifetime of a user session.
- **Supports multiple and future client types**—Extensible MIME types describe web content, so a web client can support any current and future type of downloadable content.

While the language is capable of implementing business logic (specific details and the information flow of a particular industry) in code stored and activated within the web-tier, in order to design and develop solutions that are compliant with the state standards, business logic must be decoupled from this tier and implemented within the enterprise beans tier.

Technology Guidelines

Servlets

Servlets are most effectively used for implementing logic and generating binary content. Servlets are usually not visual components, except for some that generate binary content. Instead, think of a servlet as an information service provided by an application.

A servlet can perform the services it provides (templating, security, personalization, and application control) and then select a presentation component (often a JSP page) to which it forwards the request for display.

Servlets are the preferred technology for implementing a web-tier controller, which determines how to handle a request and chooses the next view to display. A controller activates application operations and makes decisions, which are essentially procedural tasks that are best suited for program code in servlets.

JSP pages should not be used as controllers. Because JSP pages that are mostly logic are a mixture of markup tags and program code, they are difficult to read and maintain, especially for web developers who are not programmers.

Binary content should be generated by servlets. Servlets that output binary content must set the Content-Type HTTP header to the MIME type of the content being generated. JSP pages can't create binary content.

Servlets composed mostly of static text would be better implemented as JSP pages. JSP pages are for creating textual content that combines template data with dynamic data values. Servlets that print a great deal of text, and perform some logic between the print lines, are tedious to write and difficult to maintain.

JSP

JSP pages can be used for creating structured or free-form textual data. They are most appropriate where data values change between requests, but data structure either doesn't change, or changes very little. JSP pages are most appropriately used for producing structured textual content. Enterprise application data view types typically include HTML, XHTML, and DHTML.

JSP pages are best used for content that is partially fixed, with some elements that are filled in dynamically at runtime. JSP pages cannot create binary content. They are also usually not appropriate for creating content with highly variable structure or for controlling request routing. Servlets are better for those situations.

JSP pages are an excellent technology for generating XML with fixed structure. They are particularly useful for generating XML messages in standard formats, where message tags are fixed and only attribute values or character data change each time the page is served. XML documents can also be created from templates, assembling several XML subdocuments into a composite whole.

Standard tag libraries usually provide so-called "logic tags", which are custom tags that loop, perform iterations, evaluate expressions, and make decisions. Using custom tags for logic provides little benefit, and violates separation of logic and presentation.

Application Structure

The J2EE platform is a layered set of system services that are consistently available to J2EE applications across implementations. It is the top layer of a "stack" of services that support an application. The J2EE platform runs on top of the J2SE platform, which itself runs on top of the host operating system. In the web-tier, a J2EE web container provides services related to serving web requests.

Just as the J2EE platform has layers, J2EE applications can benefit from architectural layering. The highest-level division between layers in an application's web-tier is between functions that are specific to a particular application and those that occur in all web applications.

All web-tier applications share a common set of basic requirements that are not provided by the J2EE platform itself. A software layer called an application framework can meet these requirements and can be shared between applications. A web-tier application framework sits on top of the J2EE platform, providing common application functionality such as dispatching requests, invoking model methods, and selecting and assembling views. Framework classes and interfaces are structural; they are like the load-bearing elements of a building, forming the application's underpinnings. Application developers extend, use, or implement framework classes and interfaces to perform application-specific functions. A web-tier application framework makes web-tier technologies easier to use, helping application developers to concentrate on business logic.

There are a number of benefits that can be gained from choosing an existing, proven, web-tier application framework, rather than designing and building a custom framework layer. A web-tier application framework can provide the following benefits to your application:

- **Decouples presentation and logic into separate components**—Frameworks encourage separating presentation and logic because the separation is designed into the extension interfaces.
- **Separates developer roles**—Application frameworks generally provide different interfaces for different developers. Presentation component developers tend to focus on creating JSP pages using custom tags, while logic developers tend to write action classes, tag handlers, and model code. This separation allows both types of developers to work more independently.
- **Provides a central point of control**—Most frameworks provide a rich, customizable set of application-wide features, such as templating, localization, access control, and logging.
- **Facilitates unit testing and maintenance**—Because framework interfaces are consistent, automated testing harnesses are easy to build and execute.
- **Provides a rich set of features**—Adopting a framework can leverage the expertise of a group of web-tier MVC design experts. The framework may include useful features that you do not have the experience to formulate or the time to develop.
- **Encourages the development and use of standardized components**—Over time, developers and organizations can accumulate and share a toolbox of preferred components. Most frameworks incorporate a set of custom tags for view construction.
- **Provides stability**—Frameworks are usually created and actively maintained by large organizations or groups, and are used and tested in a large installed base. Accordingly, framework code tends to be more stable than custom code.
- **Has community support**—Popular frameworks attract communities of enthusiastic users who report bugs, provide consulting and training services, publish tutorials, and produce useful add-ons.

- **Reduces training costs and time**—Developers already trained and experienced in using a framework get up to speed more quickly and are more productive.
- **Supports input validation**—Many frameworks have consistent ways to specify input validation. Validation is commonly available on the client side, on the server side, or both.
- **Compatible with development tools**—Good tools can improve productivity and reliability. Some frameworks are integrated with rapid application development tool sets.

Framework Design Pattern

Model-View-Controller ("MVC") is the recommended architectural design pattern for interactive applications. MVC organizes an interactive application into three separate modules: one for the application model with its data representation and business logic, the second for views that provide data presentation and user input, and the third for a controller to dispatch requests and control flow. Most web-tier application frameworks use some variation of the MVC design pattern.

The MVC design pattern provides a host of design benefits. MVC separates design concerns (data persistence and behavior, presentation, and control), decreases code duplication, centralizes control, and makes the application more easily modifiable. MVC also helps developers with different skill sets to focus on their core skills and collaborate through clearly defined interfaces. For example, a J2EE application project may include developers of custom tags, views, application logic, database functionality, and networking. An MVC design can centralize control of such application facilities as security, logging, and screen flow. New data sources are easy to add to an MVC application by creating code that adapts the new data source to the view API. Similarly, new client types are easy to add by adapting the new client type to operate as an MVC view. MVC clearly defines the responsibilities of participating classes, making bugs easier to track down and eliminate.

A J2EE application's web-tier serves HTTP requests. At the highest level, the web-tier does four basic things in a specific order: interprets client requests, dispatches those requests to business logic, selects the next view for display, and generates and delivers the next view.

The web-tier controller receives each incoming HTTP request and invokes the requested business logic operation in the application model. Based on the results of the operation and state of the model, the controller then selects the next view to display. Finally, the controller generates the selected view and transmits it to the client for presentation. An enterprise application's web-tier commonly has the following requirements:

An application design must have a strategy for serving current and future client types.

A web-tier controller must be maintainable and extensible. Its tasks include mapping requests to application model operations, selecting and assembling views, and managing screen flow. Good structure can minimize code complexity.

Application model API design and technology selection have important implications for an application's complexity, scalability, and software quality.

Choosing an appropriate technology for generating dynamic content improves development and maintenance efficiency.

Structuring

There are two basic usage patterns for structuring the web-tier, "Model 1" and "Model 2". Model 1 and Model 2 simply refer to the absence or presence (respectively) of a controller servlet that dispatches requests from the client tier and selects views.

A Model 1 architecture consists of a web browser directly accessing web-tier JSP pages. The JSP pages access web-tier JavaBeans that represent the application model, and the next view to display (JSP page, servlet, HTML page, and so on) is determined either by hyperlinks selected in the source document or by request parameters. A Model 1 application control is decentralized, because the current page being displayed determines the next page to display. In addition, each JSP page or servlet processes its own inputs (parameters from GET or POST). In some Model 1 architectures, choosing the next page to display occurs in scriptlet code, but this usage is considered poor form.

A Model 2 architecture introduces a controller servlet between the browser and the JSP pages or servlet content being delivered. The controller centralizes the logic for dispatching requests to the next view based on the request URL, input parameters, and application state. The controller also handles view selection, which decouples JSP pages and servlets from one another. Model 2 applications are easier to maintain and extend, because views do not refer to each other directly. The Model 2 controller servlet provides a single point of control for security and logging, and often encapsulates incoming data into a form usable by the back-end MVC (Model-View-Controller) model.

The Model 2 architecture provides a framework that clearly separates the presentation, business rules, and data. This is a fundamental principle of the NCSTA.

MVC

A MVC application framework can greatly simplify implementing a Model 2 application. Application frameworks such as Apache Struts and JavaServer Faces include a configurable front controller servlet, and provide abstract classes that can be extended to handle request dispatches.

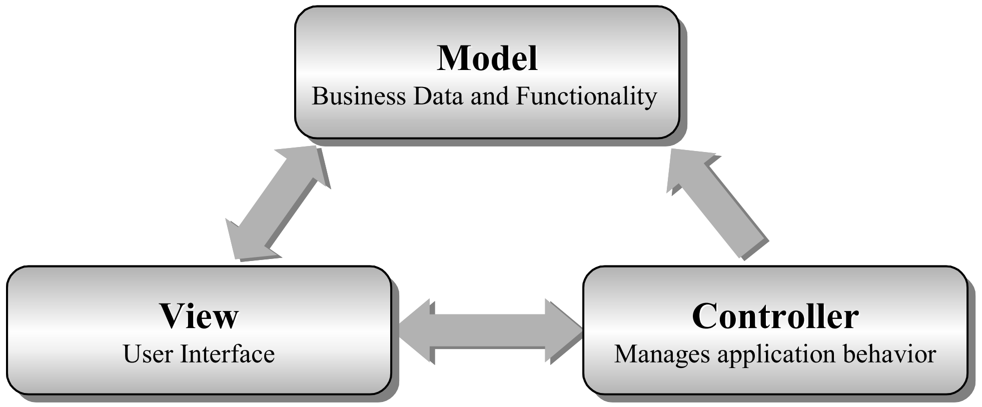

Figure 7: Model View Controller (MVC) Framework

A MVC *controller* maps incoming requests to operations on the application model, and selects views based on model and session state. web-tier controllers have a lot of duties, so they require careful design to manage complexity. Because most applications grow over time, extensibility is an important requirement.

MVC *views* display data produced by the MVC model. View components (also known as "presentation components") in the web-tier are usually JSP pages and servlets, along with such static resources as HTML pages, PDF files, graphics, and so on. JSP pages are best used for generating text-based content, often HTML or XML. Servlets are most appropriate for generating binary content or content with variable structure.

A MVC application *model* both represents business data and business logic. In accordance with the NCSTA, J2EE applications should implement their application model components as enterprise beans, which offer scalability, concurrency, load balancing, automatic resource management, and other benefits.

Separating Business Logic from Presentation

Placing business logic and presentation code in separate software layers is good design practice (and required to be compliant with the NCSTA). The business logic layer provides only application functionality, with no reference to presentation. The presentation layer presents the data and input prompts to the user (or to another system), delegating application functionality to the business logic layer.

Separating business logic from presentation has several important benefits:

- **Minimizes impact of change**—Business rules can be changed in their own layer, with little or no modification to the presentation layer. Application presentation or workflow can change without affecting code in the business layer.
- **Increases maintainability**—Most business logic occurs in more than one use case of a particular application. Business logic copied and pasted between components expresses the same business rule in two places in the application. Future changes to the rule require two edits instead of one. Business logic expressed in a separate component and accessed

referentially can be modified in one place in the source code, producing behavior changes everywhere the component is used.

- **Provides client independence and code reuse**—Intermingling data presentation and business logic ties the business logic to a particular type of client. For example, business logic implemented on client-side applications or JSP pages is not usable by a servlet or a separate application client; the code must be re-implemented for the other client types. Therefore, multiple client types can use business logic that is available referentially.
- **Separates developer roles**—Code that deals with data presentation, request processing, and business rules all at once is difficult to read, especially for a developer who may specialize in only one of these areas. Separating business logic and presentation allows developers to concentrate on their area of expertise.

The Enterprise Bean Tier

In a multi-tier J2EE application, the Enterprise JavaBeans (EJB) tier hosts application-specific business logic and provides system-level services such as transaction management, concurrency control, and security. Enterprise JavaBeans technology provides a distributed component model that enables developers to focus on solving business problems while relying on the J2EE platform to handle complex system level issues. This *separation of concerns* allows rapid development of scalable, accessible, robust, and highly secure applications. In the J2EE programming model, EJB components are a fundamental link between presentation components hosted by the web-tier (presentation) and business-critical data and systems maintained in the enterprise information system tier (data).

Business logic, in a very broad sense, is the set of procedures or methods used to manage a specific business function. Taking the object-oriented approach enables the developer to decompose a business function into a set of components or elements called *business objects*. Like other objects, business objects have both state (or data) and behavior. For example, an employee object has data such as a name, address, social security number, and so on. It has methods for assigning it to a new department or for changing its salary by a certain percentage. To manage a business problem you must be able to represent how such business objects function and interact to provide the desired functionality. The set of business-specific rules that help identify the structure and behavior of the business objects, along with the pre- and post-conditions that must be met when an object exposes its behavior to other objects in the system is known as *business logic.*

The EJB tier of the J2EE platform provides a standard server-side distributed component model that greatly simplifies the task of writing business logic. In the EJB architecture, system experts provide the framework for delivering system level services, and application domain experts provide the components that hold business-specific knowledge. The J2EE platform enables enterprise developers to concentrate on solving the problems of the enterprise instead of expending their efforts on system-level issues.

The Enterprise JavaBeans architecture defines components called *enterprise beans* that allow the developer to write business objects that use the services provided by the J2EE platform. There are three kinds of enterprise beans: session beans, entity beans, and message-driven beans.

- Session beans are intended to be private resources used only by the client that creates them. For this reason, session beans, from the client's perspective, appear anonymous.
- Entity beans are components that represent an object-oriented view of some entities that are stored in persistent storage, such as a database. In contrast to session beans, every entity bean has a unique identity that is exposed as a primary key.
- Message-driven beans are components that process asynchronous messages delivered via the Java Message Service (JMS) API. Message-driven beans, by implementing a JMS message listener interface, can asynchronously consume messages sent to a JMS queue or topic.

Enterprise beans live inside EJB containers, which provide life cycle management, transactions, security, data access, and a variety of other services for them. An EJB container is part of a J2EE application server, which provides naming and directory services, e-mail services, and so on. When a client invokes an operation on an enterprise bean, its container intercepts the call. By interposing between clients and components at the method call level, containers can inject services that propagate across calls and components, and even across containers running on different servers and different machines.

Remote and Local Component Interfaces

Within the EJB architecture, a session or entity bean has the ability to implement a local interface and/or a remote interface. An enterprise bean defines a remote client view when it is designed for use in a distributed environment; that is, when its clients may potentially reside in a different JVM. Each method call on a bean's remote interface results in a remote method invocation.

Use of a local interface avoids the performance overhead of remote method invocation. To use a local interface the enterprise bean and its client must be guaranteed to be located within the same JVM. By implementing a local interface, co-located enterprise beans can make direct, local method calls on the methods of other beans and avoid the remote invocation overhead.

Utilizing local interfaces between presentation and business logic is not the recommended approach for developing enterprise applications that are compliant with the NCSTA. However, in some cases such as EJB-to-EJB communication between different business processes, local interfaces would be acceptable.

Applications are typically distributed across a network, because client and data store resources are usually located on different machines from the application itself. An EJB-centric approach, with the business logic residing on the middle tier, gives architects the flexibility to design the application as a distributed or a local application. (Distributed applications are those that interact through remote communication mechanisms.)

The J2EE platform provides facilities to help create distributed applications, but it also lets application developers apply a local model to their application. A key consideration for developers is whether to use enterprise beans with a local or a remote interface. With careful thought, developers can use enterprise beans with both local and remote interfaces.

- Applications implemented with a local architecture model have their components reside in the same Java Virtual Machine (JVM). Because their co-location in the same address space is guaranteed, these components can communicate with each other without the overhead of remote network calls, thus permitting more efficient fine-grained access among them. As a result, these applications usually exhibit better performance.
- Applications implemented with a remote architecture model do not have a dependency to be located within the same Java Virtual Machine (JVM). Because the application can be deployed in the same or a different JVM, it is more modular, maintainable, and reusable because there is less dependency among individual components. While this approach offers greater flexibility to the application, it usually results in some decreased performance because access to a remote component involves additional overhead.

A distributed architecture is typically more scalable than a local application design. Despite the higher performance of local application architecture, its components must reside in the same address space, so it cannot scale beyond a single machine. While it is possible to partition the clients of a local application to separate instances running the same application, this becomes harder to achieve since they often need to access and update shared data. A distributed system is generally more scalable than a local application since the components are *designed from the ground up* to run in a different address space.

Distributed applications are generally more highly available than local applications. Since a distributed component does not know or depend on a particular JVM, it is easier to migrate the distributed enterprise bean component to a different JVM in the event of a hardware failure.

To provide the most scalable, accessible, adaptable, and secure system the NCSTA mandates the use of remote interfaces between the presentation and business layers. The use of local interfaces should be limited to EJB-to-EJB interaction.

Design Guidelines

Provided below are some recommended guidelines for developing EJB-tier components.

- Keep the code in enterprise beans as “client-neutral” as possible. Enterprise beans are meant to contain business logic that can be used by many client types. Methods in enterprise beans that serve only one client type make any logic within that method inaccessible to other client types. Code that is specific for a particular client type belongs with the software managing that client type. In particular, web-tier and HTTP-related functions and data do not belong in an enterprise bean.
- Keep as much business logic as possible in the EJB tier. By doing so, you take advantage of the EJB container services and simplify your programming effort.
- Utilize the EJB architecture as much as possible. The EJB architecture handles such system services as transaction management, security, scalability, persistent data access, distributed processing, and concurrency. An enterprise bean developer does not have to include code to handle these services. Instead, developers can focus on the application and business logic.

Data Access

In situations where the use of container-managed persistence is not suitable, *data access objects* can be used to encapsulate access to persistent data. A data access object (DAO) design pattern separates the interfaces to a system resource from the underlying strategy used to access that resource.

A DAO class provides an abstract API for manipulating a data source. This abstract API makes no reference to how the data source is implemented. By encapsulating data access calls, data access objects allow adapting the application’s data access to different schemas or even to different database types. DAOs share a common interface enabling the application assembler to choose the appropriate object from among several at assembly time.

The Enterprise Information System Tier

This section focuses on the integration of applications with existing data sources and applications. Enterprise Information Systems (EIS) provide the information infrastructure critical to the business processes of an enterprise. Examples of EIS include relational databases, enterprise resource planning (ERP) systems, mainframe transaction processing systems, and legacy database systems.

The EIS integration problem has assumed great importance because enterprises are striving to leverage their existing systems and resources while adopting and developing new technologies and architectures. The emergence of web-based architectures and Web Services has made it more imperative for enterprises to integrate their EIS and applications and expose them to the web.

The EIS integration problem is one part of the broader scope of Enterprise Application Integration (EAI). EAI entails integrating applications and data sources so that they can easily share business processes and data. Following are aspects of EAI, and discussions of recommended guidelines:

• **Application integration**—Existing enterprise applications may be Commercial Off-the-Shelf (COTS) bundled applications or they may be developed in-house. While such applications expose business level functionality used directly by end users or integrated with other enterprise applications, they usually do not expose the underlying data on which the business functionality is built.

• **Data integration**—Most environments contain more than one database system upon which business processes run. These database systems may be relational, object-based, hierarchical, file based, or legacy stores. Data integration focuses on integrating existing data with enterprise applications. For example, integration might entail tying a web-based order management system with an existing order and customer database.

• **Legacy integration**—Legacy integration involves integrating new applications with applications and data sources that have been in operation for some time, often referred to as "legacy" systems. Any disruption in these legacy systems is not allowed. Legacy integration focuses on how to connect applications to these legacy systems.

Integration Scenarios

A J2EE application may be configured in a number of different ways to access an enterprise information system. The following diagrams illustrate a few typical enterprise information system integration scenarios.

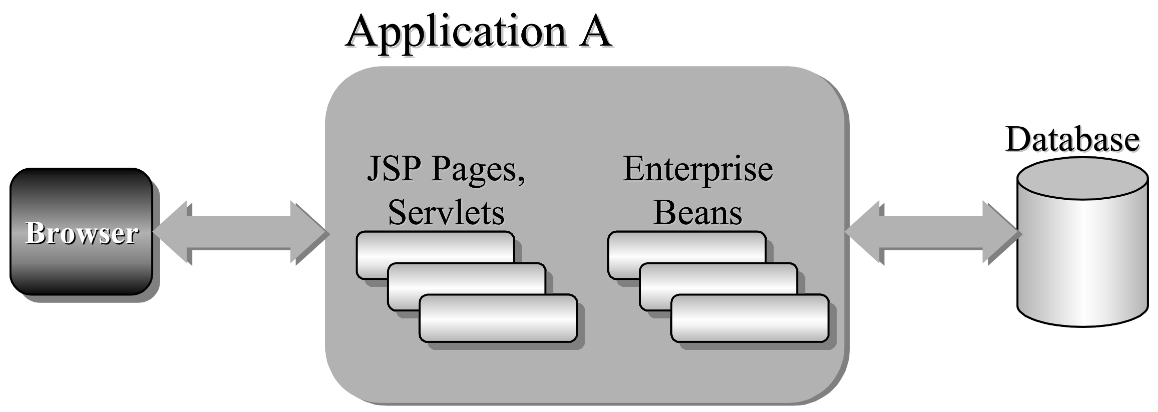

Figure 8: Basic Web-based Application Scenario

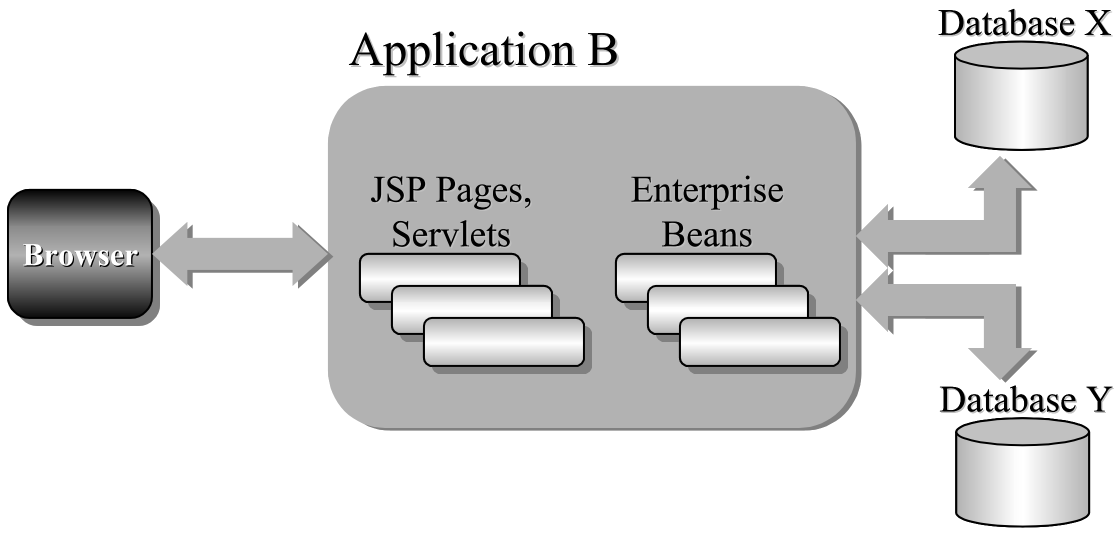

Figure 9: Web-based Data Integration Scenario

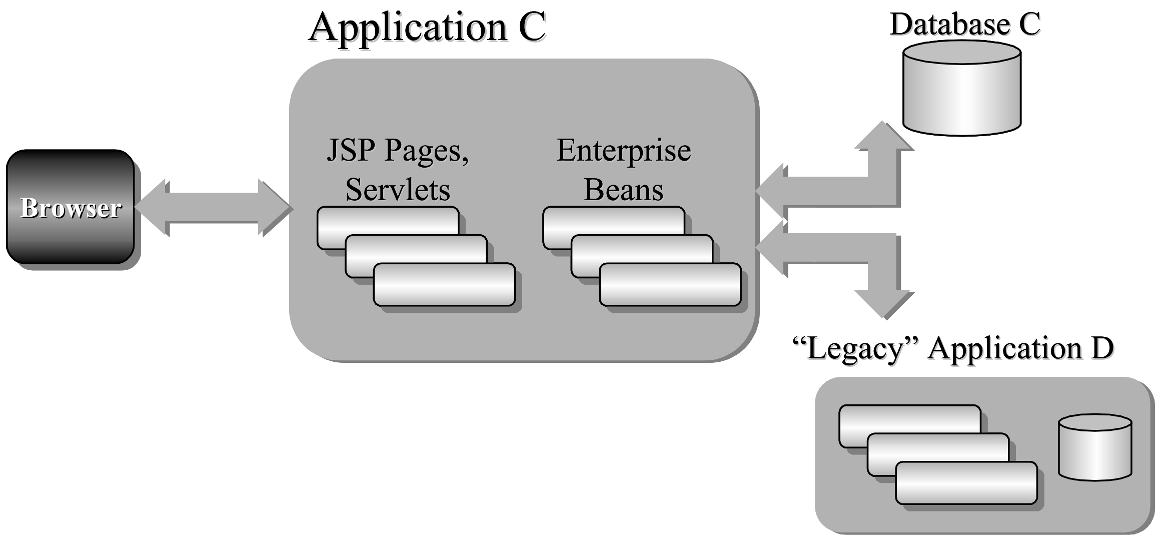

Figure 10: Web-based Legacy Integration Scenario

EIS Technologies

J2EE Connector Architecture

The J2EE Connector architecture is the standard architecture for integrating J2EE products and applications with heterogeneous enterprise information systems. The Connector architecture enables an EIS vendor to provide a standard resource adapter for its enterprise information system. Because a resource adapter conforms to the Connector specification, it can be plugged into any J2EE-compliant application server to provide the underlying infrastructure for integrating with that vendor's EIS. The EIS vendor is assured that its adapter will work with any J2EE-compliant application server. The J2EE application server, because of its support for the Connector architecture, is assured that it can connect to multiple enterprise information systems. The J2EE application server and EIS resource adapter collaborate to keep all system-level mechanisms (transactions, security, connection management) transparent to the application components. This enables an application component developer to focus on a component's business and presentation logic without getting involved in the system-level issues related to EIS integration.

Through its contracts, the J2EE Connector architecture establishes a set of programming design guidelines for EIS access. The J2EE Connector architecture defines two types of contracts: system and application level. The system-level contracts exist between a J2EE application server and a resource adapter. An application-level contract exists between an application component and a resource adapter.

The application-level contract defines the client API that an application component uses for EIS access. The Connector architecture does not require that an application component use a specific client API.

The system-level contracts define a "pluggability" standard between application servers and enterprise information systems. By developing components that adhere to these contracts, an application server and an EIS know that connecting is a straightforward operation of plugging in the resource adapter. The EIS vendor or resource adapter provider implements its side of the system-level contracts in a resource adapter, which is a system library specific to the EIS. The resource adapter is the component that plugs into an application server.

Java Message Service (JMS)

The Java Message Service (JMS) API is a standard Java API defined for enterprise messaging systems. It is a common messaging API that can be used across different types of messaging systems. A Java application uses the JMS API to connect to an enterprise messaging system. Once connected, the application uses the facilities of the underlying enterprise messaging system (through the API) to create messages and to communicate asynchronously with one or more peer applications.

A JMS engine implements the JMS API for an enterprise messaging system and provides access to the services provided by the underlying message system. Many of the industries application server vendors include a messaging engine with the application server.

The JMS API supports a peer-to-peer messaging architecture. Both the source (or producer) and the destination (or consumer) applications act as clients to the JMS engine.

A JMS domain identifies the type of asynchronous message-based communication supported by an enterprise messaging system. There are two domain types: queue-based point-to-point domains and publish-subscribe domains.

JDBC API

Relational Database Management Systems (RDBMS) are the most prevalent approach to data storage and is an industry best practice. Many application component providers use the JDBC API for accessing relational databases to manage persistent data for their applications. The JDBC API has two parts: a client API for direct use by developers to access relational databases and a standard, system-level contract between J2EE servers and JDBC drivers for supporting connection pooling and transactions.

Developers do not use the contract between J2EE servers and JDBC drivers directly. Rather, J2EE server vendors use this contract to provide pooling and transaction services to J2EE components automatically. A Java application uses the JDBC API for obtaining a database connection, retrieving database records, executing database queries and stored procedures, and performing other database functions.

In order to be compliant with the NCSTA, an application's business logic should <u>not</u> be implemented as stored procedures. Stored procedures should only be used for database specific functions.

Security

In an enterprise computing environment, failure, compromise, or lack of availability of computing resources can jeopardize the integrity of the application and/or data. Steps must be taken to identify threats to security. Once they are identified, steps should be taken to reduce these threats.

The J2EE application-programming model attempts to leverage existing security services rather than require new services or mechanisms.

Security Threats and Mechanisms

Threats to critical assets fall into a few general categories:

- Disclosure of confidential information
- Modification or destruction of information
- Misappropriation of protected resources
- Compromise of accountability
- Misappropriation that compromises availability

Depending on the environment in which an application operates, these threats may manifest in different forms. For example, in a traditional single system environment, a threat of disclosure might become apparent in the vulnerability of information kept in files. In a distributed environment with multiple servers and clients, a threat of disclosure might also result from exposures occurring as the result of networking.

Although not all potential threats can or need to be eliminated, there are many circumstances where exposure can be reduced to an acceptable level through the use of the following security mechanisms: authentication, authorization, signing, encryption, and auditing.

All projects are required to complete a security risk assessment. Through a proper security risk assessment, potential security exposures are managed to a risk level commensurate with the criticality and sensitivity of the application and associated data.

Authentication

In distributed component computing, *authentication* is the mechanism by which users and service providers prove to one another that they are acting on behalf of specific users or systems. When the proof is bi-directional, it is referred to as *mutual authentication.* Authentication establishes the call identities and proves that the participants are authentic instances of these identities. An entity that participates in a call without establishing or proving an identity (that is, *anonymously*) is called *unauthenticated*.

When a client *program,* run by a user, makes the calls, the caller identity is likely to be that of the *user*. When the caller is an *application component* acting as an intermediary in a call chain originating with some user, the identity may be associated with that of the user, in which case the component would be *impersonating* the user. Alternatively, one application component may call another with an identity of its own and unrelated to that of its caller.

Authentication is often achieved in two phases. First, an *authentication context* is established by performing a service-independent authentication requiring knowledge of some secret. The authentication context encapsulates the identity and is able to fabricate *authenticators* (proofs of identity*)*. Then, the authentication context is used to authenticate with other (called or calling) entities.

The basis of authentication entails controlling access to the authentication context and thus the ability to authenticate as the associated identity. Among the possible policies and mechanisms for controlling access to an authentication context are:

- Once the user performs an initial authentication, the processes the user starts inherit access to the authentication context.
- When a component is authenticated, access to the authentication context may be available to other related or trusted components, such as those that are part of the same application.
- When a component is expected to *impersonate* its caller, the caller may *delegate* its authentication context to the called component.

Authorization

Authorization mechanisms limit interactions with resources to collections of users or systems for the purpose of enforcing integrity, confidentiality, or availability constraints. Such mechanisms allow only authentic caller identities to access components.

Mechanisms provided by the J2EE platform can be used to control access to code based on identity properties, such as the location and signer of the calling code, and the identity of the user of the calling code. A credential is made available to the called component. The credential contains information describing the caller through its identity attributes. In the case of anonymous callers, a special credential is used. These attributes uniquely identify the caller in the context of the authority that issued the credential.

Depending on the type of credential, it may also contain other attributes that define shared authorization properties, such as group memberships, that distinguish collections of related credentials. The identity and shared authorization attributes in the credential are referred to as the caller's *security attributes.* In the J2SE platform, the identity attributes of the code used by the caller may also be included in the caller's security attributes. Access to the called component is determined by comparing the caller's security attributes with those required to access the called component.

In the J2EE architecture, a container serves as an authorization boundary between the components it hosts and their callers. The authorization boundary exists inside the container's authentication boundary so that authorization is considered in the context of successful authentication. For inbound calls, the container compares security attributes from the caller's credential with the access control rules for the target component. If the rules are satisfied, the call is allowed. Otherwise, the call is rejected.

There are two fundamental approaches to defining access control rules: *capabilities* and *permissions*. Capabilities focus on what a caller can do. Permissions focus on who can do something. The J2EE application-programming model focuses on permissions.

The State of North Carolina

The Identity and Access Management Service (IAMS) provides state agencies with a common centralized shared service to authenticate, authorize, and manage the identities of users as well as provide audit logging of user activity, in a secure and consistent manner. Agencies are required to utilize this service for all in-house developed applications.

IAMS is a Commercial-Off-The-Shelf (COTS) solution, which provides customizable user provisioning, in addition to the traditional authentication and authorization functionality of most directory based technologies. Implementations may be customized to meet agency business requirements, as shown in the sample workflow in Figure 11.

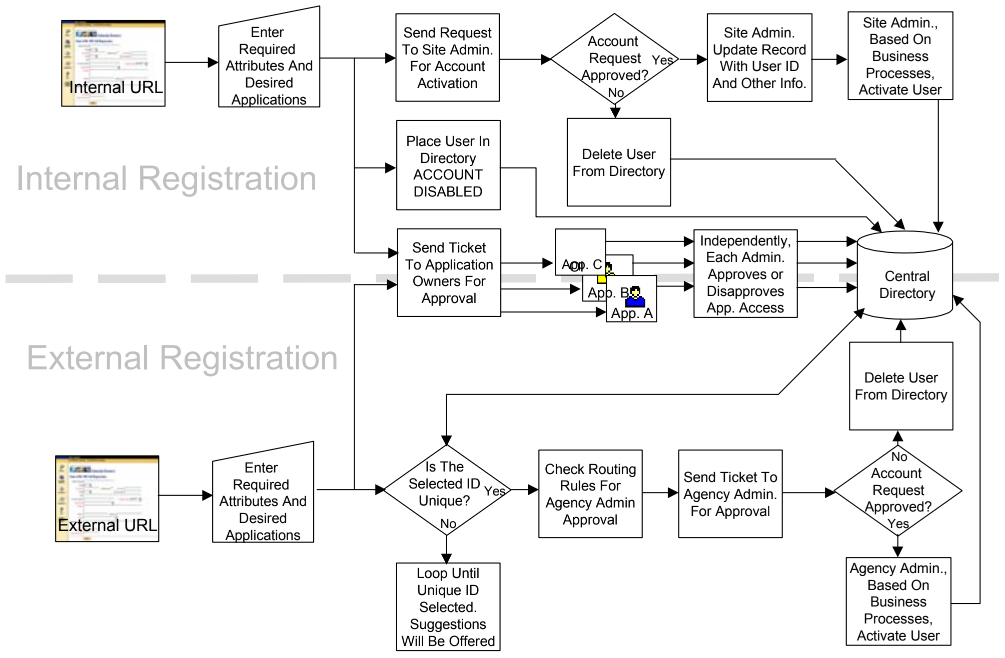
Internal URL
Enter Required Attributes And Desired Applications
Send Request To Site Admin. For Account Activation
Account Request Approved?
Yes
No
Site Admin. Update Record With User ID And Other Info.
Site Admin., Based On Business Processes, Activate User
Place User In Directory ACCOUNT DISABLED
Delete User From Directory
Internal Registration
Send Ticket To Application Owners For Approval
App. C
App. B
App. A
Independently, Each Admin. Approves or Disapproves App. Access
Central Directory
External Registration
Delete User From Directory
External URL
Enter Required Attributes And Desired Applications
Is The Selected ID Unique?
Yes
No
Check Routing Rules For Agency Admin Approval
Send Ticket To Agency Admin. For Approval
No
Account Request Approved?
Yes
Loop Until Unique ID Selected. Suggestions Will Be Offered
Agency Admin., Based On Business Processes, Activate User

Figure 11: Example of a customized workflow

The IAMS solution provides for a robust Application Programming Interface (API) based on XML or a service loaded on a web server that utilizes HTTP header variables. Agency individual needs will dictate the appropriate approach.

IAMS is structured as a central service, providing authentication to validate user credentials (Figure 12) and authorization (access control) to the resource requested by the user. More granular authorization is retained at the application and is role-based. Roles are defined, typically in the system database, which match roles defined in IAMS. IAMS will pass role information to the application through the API or HTTP header variables. The application then handles the role information to further control access to functions within the application on a more granular level.

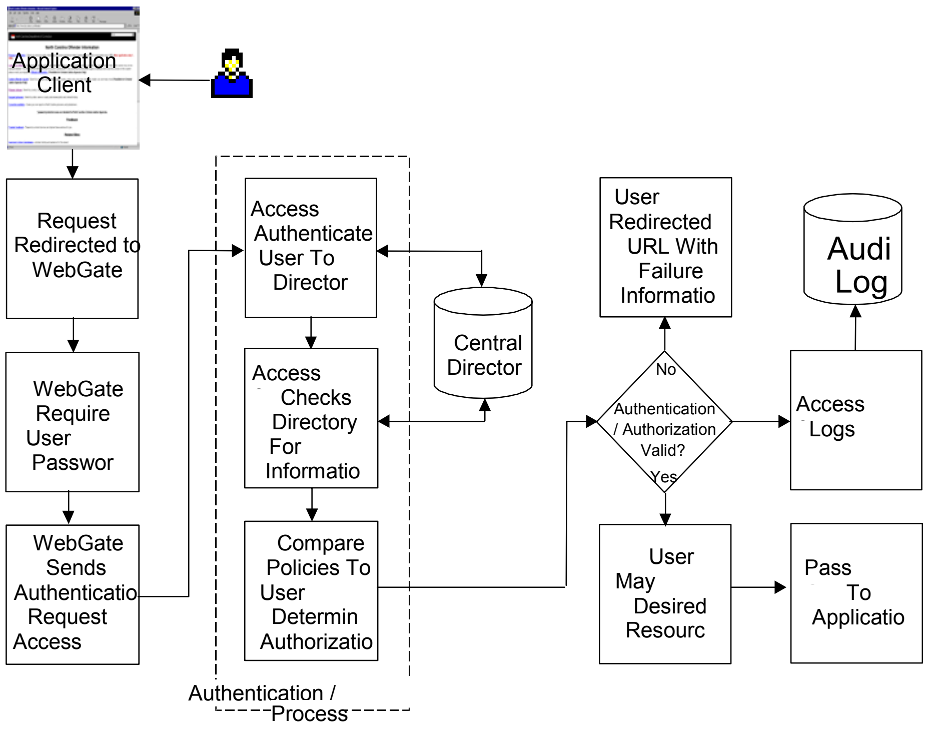
Application Client
Request Redirected to WebGate
WebGate Require User Passwor
WebGate Sends Authenticatio Request Access
Access Authenticate User To Director
Access Checks Directory For Informatio
Compare Policies To User Determin Authorizatio
Central Director
Authentication / Process
User Redirected URL With Failure Informatio
No
Authentication / Authorization Valid?
Yes
User May Desired Resourc
Audi Log
Access Logs
Pass To Applicatio

Figure 12: IAMS Authentication Process

Signing

In a distributed computing system, a significant amount of information is transmitted through networks in the form of messages. Message content is subject to three main types of attacks. Messages might be intercepted and modified for the purpose of changing the effect they have on their recipients. Messages might be captured and reused one or more times for the benefit of another party. An eavesdropper might monitor messages in an effort to capture information that would not otherwise be available. Using integrity and confidentiality mechanisms can minimize such attacks.

Integrity Mechanisms

Integrity mechanisms ensure that another party, especially one that can intercept and modify their communications, is not tampering with communication between entities. Integrity mechanisms can also be used to ensure that messages can only be used once.

Attaching a message signature to a message ensures message integrity. The message signature is calculated by using a one-way hash algorithm to convert the message contents into a typically smaller, fixed-length *message digest* that is then *signed.* A message signature ensures that modification of the message by anyone other than the caller will be detectable by the receiver.

Confidentiality Mechanisms

Confidentiality mechanisms ensure private communication between entities. Privacy is achieved by encrypting the message contents. Symmetric (or shared secret) encryption mechanisms generally require less computing resources than asymmetric (or public key) mechanisms. It is therefore quite common to use an asymmetric mechanism to secure the exchange of a symmetric encryption key that is then used to encrypt the message traffic.

Definition of Terms

This section contains definitions of common Java and J2EE related terms[2].

API - Application Programming Interface. The specification of how a programmer writing an application accesses the behavior and state of classes and objects.

Business logic - The code that implements the functionality of an application. In J2EE, this logic is implemented by the methods of an enterprise bean.

Container - An entity that provides life cycle management, security, deployment, and runtime services to components. Each type of container (EJB, Web, JSP, servlet, applet, and application client) also provides component-specific services.

CORBA - Common Object Request Broker Architecture. A language independent, distributed object model specified by the Object Management Group (OMG).

Distributed application - An application made up of distinct components running in separate runtime environments, usually on different platforms connected through a network. Typical distributed applications are 3-tier (client/middleware/server), and n-tier (client/multiple middleware/multiple servers).

DOM - Document Object Model. A tree of objects with interfaces for traversing the tree and writing an XML version of it, as defined by the W3C specification.

EJB - Enterprise JavaBeans. - A component architecture for the development and deployment of object-oriented, distributed, enterprise-level applications. Applications written using the Enterprise JavaBeans architecture are scalable, transactional, and multi-user and secure.

EJB container - A container that implements the EJB component contract of the J2EE architecture. This contract specifies a runtime environment for enterprise beans that includes security, concurrency, life cycle management, transaction, deployment, and other services. An EJB container is provided by an EJB or J2EE application server.

EIS - Enterprise Information System. The applications that comprise an enterprise's existing system for handling company-wide information. These applications provide an information infrastructure for an enterprise. An EIS offers a well defined set of services to its clients. These services are exposed to clients as local and/or remote interfaces. Examples of EISs include: an ERP system, a mainframe transaction processing system, or a database system.

Entity bean - An enterprise bean that represents persistent data maintained in a database. An entity bean can manage its own persistence or it can delegate this function to its container.

GUI - Graphical User Interface. Refers to the techniques involved in using graphics, along with a keyboard and a mouse, to provide an easy-to-use interface to some program.

HTML - HyperText Markup Language. This is a file format, based on SGML, for hypertext documents on the Internet.

HTTP - HyperText Transfer Protocol. The Internet protocol, based on TCP/IP, used to fetch hypertext objects from remote hosts.

HTTPS - HyperText Transfer Protocol layered over the SSL protocol.

IIOP - Internet Inter-ORB Protocol. A protocol used for communication between CORBA object request brokers.

IP - Internet Protocol. The basic protocol of the Internet. It enables the unreliable delivery of individual packets from one host to another. It makes no guarantees about whether or not the packet will be delivered, how long it will take, or

[2] Glossary of Java Related Terms provided by http://java.sun.com/docs/glossary.html

if multiple packets will arrive in the order they were sent. Protocols built on top of this add the notions of connection and reliability. See TCP/IP.

JAX-RPC - Java API for XML-based Remote Procedure Calls. Enables Java technology developers to develop SOAP based interoperable and portable web services. JAX-RPC provides the core API for developing and deploying web services messaging on the Java platform.

JDBC - Java Database Connectivity. An industry standard for database-independent connectivity between the Java platform and a wide range of databases. The JDBC interface provides a call-level API for SQL-based database access.

JFC - Java Foundation Classes. An extension that adds graphical user interface class libraries to the Abstract Windowing Toolkit (AWT).

JMS - Java Message Service. An API for using enterprise messaging systems such as IBM MQ Series, TIBCO Rendezvous, MSMQ, and so on.

JNDI - Java Naming and Directory Interface. A set of APIs that assists with the interfacing to multiple naming and directory services.

RMI - Java Remote Method Invocation. A distributed object model for Java program to Java program, in which the methods of remote objects written in the Java programming language can be invoked from other Java virtual machines, possibly on different hosts.

Java Technologies - A set of technologies that enable the creation and safe running of software programs in both stand-alone and networked environments.

JTA - Java Transaction API. An API that allows applications and J2EE servers to access transactions.

JVM - Java virtual machine. A software "execution engine" that safely and compatibly executes the byte codes in Java class files on a microprocessor (whether in a computer or in another electronic device).

JavaMail - An API for sending and receiving email.

JSP - JavaServer Pages. An extensible web technology that uses template data, custom elements, scripting languages, and server-side Java objects to return dynamic content to a client. Typically the template data is HTML or XML elements, and in many cases the client is a web browser.

JAXP – Java API for XML Processing. Enables applications to parse and transform XML documents using an API that is independent of a particular XML processor implementation. JAXP also provides a feature, which enables applications to easily switch between particular XML processor implementations.

JDBC - Java Database Connectivity. An industry standard for database-independent connectivity between the Java platform and a wide range of databases. The JDBC interface provides a call-level API for SQL-based database access.

JSP container - A container that provides the same services as a servlet container and an engine that interprets and processes JSP pages into a servlet.

ORB - Object Request Broker. A library than enables CORBA objects to locate and communicate with one another.

RMI - Java Remote Method Invocation. A distributed object model for Java program to Java program, in which the methods of remote objects written in the Java programming language can be invoked from other Java virtual machines, possibly on different hosts.

RPC - Remote Procedure Call. Executing what looks like a normal procedure call (or method invocation) by sending network packets to some remote host.

SAX - Simple API for XML. An event-driven, serial-access mechanism for accessing XML documents.

SSL - Secure Socket Layer. A widely used means for securely communicating between a web browser and web server. SSL works by using a public key to encrypt data that's transferred over the SSL connection.

Servlet - A Java program that extends the functionality of a web server, generating dynamic content and interacting with web clients using a request-response paradigm.

Servlet container - A container that provides the network services over which requests and responses are sent, decodes requests, and formats responses. All servlet containers must support HTTP as a protocol for requests and responses, but may also support additional request-response protocols such as HTTPS.

Session bean - An enterprise bean that is created by a client and that usually exists only for the duration of a single client/server session. A session bean performs operations, such as calculations or accessing a database, for the client. While a session bean may be transactional, it is not recoverable should a system crash occur. Session bean objects can be either stateless or they can maintain conversational state across methods and transactions. If they do maintain state, then the EJB container manages this state if the object must be removed from memory. However, the session bean object itself must manage its own persistent data.

SOAP - The Simple Object Access Protocol. Uses a combination of XML-based data structuring and the Hyper Text Transfer Protocol (HTTP) to define a standardized method for invoking methods in objects distributed in diverse operating environments across the Internet.

SQL - Structured Query Language. The standardized relational database language for defining database objects and manipulating data.

TCP/IP - Transmission Control Protocol based on IP. This is an Internet protocol that provides for the reliable delivery of streams of data from one host to another.

Thin Client - A system that runs a very light operating system with no local system administration and executes applications delivered over the network.

Web container - A container that implements the web component contract of the J2EE architecture. This contract specifies a runtime environment for web components that includes security, concurrency, life cycle management, transaction, deployment, and other services. A container that provides the same services as a JSP container and federated view of the J2EE platform APIs. A web container is provided by a Web or J2EE server.

Web server - Software that provides services to access the Internet, an intranet, or an extranet. A web server hosts web sites, provides support for HTTP and other protocols, and executes server-side programs (such as servlets) that perform certain functions. In the J2EE architecture, a web server provides services to a web container. For example, a web container typically relies on a web server to provide HTTP message handling.

Web Services - Web Services are loosely coupled software components capable of collaborating with each other over multiple networks to deliver a specific result to an end user. In the process, they leverage an emerging group of standards that govern their description and interaction, including SOAP (Simple Object Access Protocol), UDDI (Universal Discovery and Description Initiative), XML (Extensible Markup Language), and WSDL (Web Services Description Language).

WSDL - The Web Services Description Language (WSDL) is an XML language that is used to describe a web service and to specify how to communicate with the web service.

XML - Extensible Markup Language. A markup language that allows you to define the tags (markup) needed to identify the data and text in XML documents. J2EE deployment descriptors are expressed in XML.

Acknowledgements

Thanks to Sun Microsystems, Inc. for providing much of the content of this guide, which was adapted from the Java BluePrints program.

Specifically, the information contained in this document is an overview of key concepts from the ***Designing Enterprise Applications with the J2EE^{TM} Platform, Second Edition (2002)*** authored by Inderjeet Singh, Beth Stearns, Mark Johnson, and the Sun Microsystems' Enterprise Team. (Sun Microsystems, Inc. 901 San Antonio Road, Palo Alto, California 94303 U.S.A.)

…and to a number of Sun Microsystems' groups for providing a review of this guide.

State and Local Government Group

Java Center of Excellence

Software Products Group

In addition, several North Carolina state agencies have reviewed and provided feedback on this document.

Office of Information Technology Services

Office of the State Controller

Department of Administration

Department of Health and Human Services

Department of Environment and Natural Resources

Department of Justice